Créer et modifier une table : CREATE et ALTER

L'école veut enregistrer ses professeurs, mais aucune table n'existe pour eux. Tu écris toi-même la structure de la table en SQL, avec ses types et ses contraintes, puis tu la fais évoluer.

4TTR 5TTR 6TTR Découverte

La base de l'école contient déjà ses élèves. Le directeur veut maintenant y enregistrer les professeurs, mais aucune table n'est prévue pour eux. Cette fois, pas de fenêtre à remplir : tu vas écrire la structure de la table toi-même, en SQL.

> 💡 BASE DE DÉPART
>
> Télécharge `mon-ecole-eleves.db` dans la barre latérale et ouvre-la avec DB Browser for SQLite. Elle contient une seule table, `eleve`, avec 12 élèves. Tout le SQL de cet article s'écrit dans l'onglet **Exécuter le SQL**.

🎯 Objectifs

À la fin de cet article, tu seras capable de :

1. **Distinguer** une instruction qui définit la structure (DDL) d'une instruction qui manipule les lignes (DML).
2. **Écrire** un `CREATE TABLE` sans erreur de syntaxe.
3. **Choisir** le type de chaque colonne : `TEXT`, `INTEGER` ou `REAL`.
4. **Protéger** les données avec des contraintes : `PRIMARY KEY`, `NOT NULL`, `UNIQUE`, `DEFAULT`.
5. **Faire évoluer** une table existante avec `ALTER TABLE`.
6. **Enregistrer** un script SQL et le rejouer, en connaissant son danger.

Deux familles d'instructions SQL

Pour lire une table, on écrit un `SELECT`. Cette instruction affiche des lignes, mais elle ne peut pas créer une table : la table doit exister avant. SQL contient donc d'autres instructions, qui se rangent en deux familles.

Famille	Ce qu'elle fait	Instructions
DML	lire et changer **les lignes** d'une table	`SELECT`, `INSERT`, `UPDATE`, `DELETE`
DDL	créer et changer **la structure** d'une table	`CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`

> 📖 NOUVELLE NOTION : LE DML

Le **DML** regroupe les instructions qui travaillent sur les lignes d'une table : les lire (`SELECT`), en ajouter (`INSERT`), les modifier (`UPDATE`) et les supprimer (`DELETE`). La table elle-même ne change pas de forme.

En anglais : GB ***Data Manipulation Language***, le langage de manipulation des données.

📖 **NOUVELLE NOTION : LE DDL**

Le **DDL** regroupe les instructions qui définissent la structure de la base : créer une table (`CREATE TABLE`), la modifier (`ALTER TABLE`) ou la supprimer (`DROP TABLE`). Elles décident quelles colonnes existent, pas ce qu'elles contiennent.

En anglais : GB ***Data Definition Language***, le langage de définition des données.

🧠 `SELECT` fait partie du DML. Il ne modifie rien, mais il travaille sur les lignes : c'est de la manipulation de données, en lecture seule.

La structure d'une base : le schéma

Ouvre l'onglet **Structure de la base**. Tu y vois la table `eleve`, ses colonnes et leur type. Déplie la table : DB Browser montre aussi l'instruction qui l'a créée.

```sql
CREATE TABLE eleve (
    id              INTEGER PRIMARY KEY,
    nom             TEXT NOT NULL,
    prenom          TEXT NOT NULL,
    date_naissance TEXT,
    commune         TEXT,
    email           TEXT UNIQUE,
    absences        INTEGER
)
```

Cette instruction ne contient aucun élève. Elle décrit seulement la forme de la table : le nom des colonnes, leur type et quelques règles.

📖 **NOUVELLE NOTION : LE SCHÉMA**

Le **schéma** d'une base, c'est sa structure : la liste de ses tables, de leurs colonnes, de leurs types et de leurs contraintes. Il ne contient aucune donnée. Le schéma s'écrit en DDL.

En anglais : GB ***schema***.

Pour ajouter les professeurs, il faut donc agrandir le schéma avec une nouvelle table.

Le besoin : une table pour les professeurs

Le directeur précise sa demande :

> « Pour chaque professeur, je veux son nom, son prénom et son adresse e-mail professionnelle. Deux professeurs ne peuvent pas avoir la même adresse. Je veux aussi savoir s'il travaille encore à l'école. »

On en tire les colonnes de la table `professeur` :

Colonne	Contenu	Règle
`id`	un numéro qui désigne le professeur	se remplit tout seul
`nom`	le nom	obligatoire
`prenom`	le prénom	obligatoire
`email`	l'adresse e-mail	obligatoire, jamais deux fois la même
`actif`	`1` s'il travaille à l'école, `0` sinon	vaut `1` si on ne précise rien

Écrire le CREATE TABLE

Une première tentative

Dans l'onglet **Exécuter le SQL**, tape exactement ceci, puis exécute-le. Une erreur s'y cache.

```sql
CREATE TABLE professeur (
    id        INTEGER PRIMARY KEY,
    nom       TEXT    NOT NULL,
    prenom    TEXT    NOT NULL,
    email     TEXT    NOT NULL UNIQUE,
    actif     INTEGER NOT NULL DEFAULT 1,
);
```

SQLite refuse :

```
near ")": syntax error
```

Le message veut dire « erreur de syntaxe près de `)` ». La virgule après `DEFAULT 1` annonce une colonne de plus. SQLite attend donc un nom de colonne, et tombe sur la parenthèse fermante.

L'erreur inverse

Autre essai, cette fois sans aucune virgule :

```sql
CREATE TABLE professeur (
    id        INTEGER PRIMARY KEY
    nom       TEXT    NOT NULL
    prenom    TEXT    NOT NULL
);
```

```
near "nom": syntax error
```

Sans virgule, SQLite croit que `nom` fait encore partie de la description de `id`, et ne comprend plus rien.

> 🧠 **UNE VIRGULE ENTRE LES COLONNES, JAMAIS APRÈS LA DERNIÈRE.** Quand un `CREATE TABLE` est refusé, regarde le mot cité après `near` : l'erreur se trouve juste avant lui.

La bonne version

```sql
CREATE TABLE professeur (
    id        INTEGER PRIMARY KEY,
    nom       TEXT    NOT NULL,
    prenom    TEXT    NOT NULL,
    email     TEXT    NOT NULL UNIQUE,
    actif     INTEGER NOT NULL DEFAULT 1
);
```

Cette fois, l'instruction passe. Vérifie avec un `SELECT` :

```sql
SELECT * FROM professeur;
```

```
id | nom | prenom | email | actif
---+-----+--------+-------+------
```

La table existe, avec ses cinq colonnes. Elle ne contient encore aucune ligne.

> 💡 Les espaces qui alignent les types et les contraintes ne sont pas obligatoires. Ils rendent seulement l'instruction plus facile à relire.

Lire un CREATE TABLE morceau par morceau

`CREATE TABLE` veut dire littéralement *crée la table*. Vient ensuite le nom de la table, puis, entre parenthèses, la liste de ses colonnes. Le point-virgule termine l'instruction.

```sql
CREATE TABLE professeur (      ← crée la table « professeur »
    id    INTEGER PRIMARY KEY, ← une colonne : nom, type, contraintes
    ...
    actif INTEGER NOT NULL DEFAULT 1   ← dernière colonne : pas de virgule
);                             ← fin de l'instruction
```

Chaque colonne se décrit toujours dans le même ordre :

```
nom_de_la_colonne   TYPE   contraintes éventuelles
```

Le type dit quelle sorte de valeur la colonne contient. Les contraintes ajoutent des règles. Les deux sections suivantes les détaillent.

Choisir le type de chaque colonne

SQLite propose trois types pour ce qu'on stocke habituellement.

Type	Traduction	Pour quoi	Exemples
`TEXT`	texte	des mots, des codes, des dates au format `AAAA-MM-JJ`	`'Dubois'`, `'2010-03-12'`
`INTEGER`	entier	des nombres sans virgule qu'on compte ou compare	`17`, `0`, `1`
`REAL`	réel	des nombres à décimales	`13.5`, `3.99`

⚠️ Dans un nombre à décimales, SQL utilise un **POINT** : `13.5`, jamais `13,5`.

Une donnée composée de chiffres n'est pas forcément un nombre

Un numéro de téléphone ne contient que des chiffres. Est-ce un `INTEGER` ? Demande à SQLite de transformer un numéro en entier. `CAST(… AS INTEGER)` veut dire *convertis … en entier* ; tu n'as pas besoin de le retenir.

```
SELECT CAST('0471234567' AS INTEGER) AS telephone_en_nombre;
```

```
telephone_en_nombre
-------------------
471234567
```

Le zéro du début a disparu : le numéro est devenu faux. Et personne n'additionne jamais deux numéros de téléphone. Un téléphone se range donc dans une colonne `TEXT`, comme un code postal, un numéro de compte ou un code-barres.

🧠 **AVANT DE CHOISIR `INTEGER`, DEMANDE-TOI SI TU VAS CALCULER AVEC CETTE VALEUR.** Si la réponse est non, c'est probablement du `TEXT`.

La clé primaire

La colonne `id` porte la mention `INTEGER PRIMARY KEY`. Elle donne à chaque professeur un numéro que personne d'autre ne peut avoir.

> 📖 **NOUVELLE NOTION : LA CLÉ PRIMAIRE**
>
> La **clé primaire** est la colonne qui désigne une ligne et une seule. Deux lignes ne peuvent jamais avoir la même valeur dans cette colonne, et elle ne peut pas rester vide. Chaque table a une seule clé primaire.
>
> En anglais : GB ***primary key***.

En SQLite, une colonne déclarée exactement `INTEGER PRIMARY KEY` **se remplit toute seule** : si tu ne donnes pas de numéro, SQLite prend le plus grand numéro existant et ajoute 1.

> 💡 Sur le web, tu croiseras souvent le mot `AUTOINCREMENT`, et DB Browser propose une case **AI**. Avec SQLite, c'est inutile : `INTEGER PRIMARY KEY` suffit. N'écris pas `AUTOINCREMENT`.

Les contraintes : des règles que la base fait respecter

L'absence de valeur : NULL

Regarde les adresses e-mail des élèves : trois élèves n'en ont pas. Dans ces cases, DB Browser affiche `NULL`.

> 📖 **NOUVELLE NOTION : NULL**
>
> `NULL` signifie « aucune valeur ». Ce n'est ni un texte vide, ni le nombre zéro : c'est une case que personne n'a remplie. Une colonne sans règle particulière accepte `NULL`.
>
> En anglais : GB ***null***, qui veut dire *nul, inexistant*.

Pour un professeur, un nom manquant n'a aucun sens. Il faut que la base elle-même refuse cette situation.

> 📖 **NOUVELLE NOTION : LA CONTRAINTE**
>
> Une **contrainte** est une règle écrite dans la structure d'une table. La base vérifie cette règle à chaque ajout ou modification, et refuse toute ligne qui ne la respecte pas. Elle protège les données même quand quelqu'un se trompe.
>
> En anglais : GB ***constraint***.

Rendre une colonne obligatoire : NOT NULL

`NOT NULL` se traduit par *pas nul* : la colonne ne peut pas rester vide.

> 📖 **NOUVELLE NOTION : NOT NULL**
>
> La contrainte `NOT NULL` rend une colonne obligatoire. Toute ligne dont cette colonne vaudrait `NULL` est refusée par la base.
>
> En anglais : GB ***not null***, *pas nul*.

Dans `professeur`, `nom`, `prenom`, `email` et `actif` sont `NOT NULL`. Pour chaque colonne, pose-toi la question : **une ligne a-t-elle encore du sens si cette information manque ?** Si la réponse est non, ajoute `NOT NULL`.

Interdire les doublons : UNIQUE

`UNIQUE` veut dire *unique* : une même valeur ne peut apparaître qu'une fois dans la colonne.

> 📖 **NOUVELLE NOTION : UNIQUE**
>
> La contrainte `UNIQUE` interdit les doublons dans une colonne. Si une valeur existe déjà, une deuxième ligne avec la même valeur est refusée.
>
> En anglais : GB ***unique***.

> ⚠️ **`UNIQUE` ACCEPTE PLUSIEURS `NULL`.** La colonne `eleve.email` est `UNIQUE`, et pourtant trois élèves n'ont pas d'adresse. Pour SQLite, deux valeurs inconnues ne sont pas égales. Si tu veux « pas de doublon » **et** « toujours rempli », il faut les deux : `NOT NULL UNIQUE`, comme pour `professeur.email`.

Prévoir une valeur par défaut : DEFAULT

La plupart des professeurs enregistrés travaillent à l'école. Plutôt que d'écrire `1` à chaque fois, on le prévoit dans la structure : `DEFAULT 1` veut dire *par défaut, 1*.

> 📖 **NOUVELLE NOTION : LA VALEUR PAR DÉFAUT**
>
> La **valeur par défaut** d'une colonne est la valeur que la base écrit elle-même quand on ajoute une ligne sans remplir cette colonne. Elle se déclare avec `DEFAULT` suivi de la valeur.
>
> En anglais : GB ***default value***.

Récapitulatif des contraintes

Contrainte	Traduction	Effet	Dans `professeur`
`PRIMARY KEY`	clé primaire	désigne la ligne, jamais deux fois la même valeur	`id`
`NOT NULL`	pas nul	la colonne est obligatoire	`nom`, `prenom`, `email`, `actif`
`UNIQUE`	unique	pas de doublon (mais plusieurs `NULL` possibles)	`email`
`DEFAULT`	par défaut	valeur écrite quand on n'en donne pas	`actif` (`1`)

Tu verras ces contraintes refuser des lignes dès que tu commenceras à remplir la table.

Créer ou supprimer sans erreur

Exécuter deux fois le même CREATE TABLE

Relance le `CREATE TABLE professeur` que tu viens d'exécuter. SQLite refuse :

```
table professeur already exists
```

Le message dit « la table professeur existe déjà ». On peut demander à SQLite de ne créer la table que si elle n'existe pas encore. `IF NOT EXISTS` veut dire *si elle n'existe pas* :

```sql
CREATE TABLE IF NOT EXISTS professeur (
    id        INTEGER PRIMARY KEY,
    nom       TEXT    NOT NULL,
    prenom    TEXT    NOT NULL,
    email     TEXT    NOT NULL UNIQUE,
    actif     INTEGER NOT NULL DEFAULT 1
);
```

Cette fois, aucune erreur. Et rien ne change : la table existait, SQLite l'a laissée telle quelle.

Supprimer une table : DROP TABLE

`DROP TABLE` veut dire *laisse tomber la table* : l'instruction la supprime entièrement.

```sql
DROP TABLE professeur;
```

Si la table n'existe pas, SQLite répond par une erreur qui commence par `no such table` (« pas de table de ce nom »). Pour éviter cette erreur, on ajoute `IF EXISTS`, *si elle existe* :

```sql
DROP TABLE IF EXISTS professeur;
```

> ⚠️ `DROP TABLE` **DÉTRUIT LA TABLE ET TOUTES SES LIGNES.** Il ne demande aucune confirmation. Sur une table qui contient de vraies données, c'est une catastrophe.

Si tu viens d'essayer ces instructions, recrée la table avec la bonne version du `CREATE TABLE` avant de continuer.

Modifier une table qui existe : ALTER TABLE

Le secrétariat ajoute une demande : il veut aussi le numéro de téléphone des professeurs. Supprimer et recréer la table fonctionnerait tant qu'elle est vide. Mais le jour où elle contiendra des professeurs, `DROP TABLE` les effacerait tous. Il faut pouvoir modifier la table **sans la détruire**.

Ajouter une colonne

`ALTER TABLE` veut dire *modifie la table*, et `ADD COLUMN` *ajoute la colonne*.

```
ALTER TABLE professeur ADD COLUMN telephone TEXT;
```

```
SELECT * FROM professeur;
```

```
id | nom | prenom | email | actif | telephone
---+-----+--------+-------+-------+----------
```

La colonne `telephone` arrive en dernière position. Elle est de type `TEXT`, pour garder le zéro du début. Si la table avait contenu des lignes, elles auraient toutes reçu `NULL` dans cette nouvelle colonne.

> 📖 **NOUVELLE NOTION : ALTER TABLE**
>
> L'instruction `ALTER TABLE` modifie la structure d'une table qui existe déjà, sans effacer ses lignes. Avec SQLite, elle sert surtout à ajouter une colonne, à renommer une colonne ou à renommer la table.
>
> En anglais : GB ***alter***, *modifier*.

Renommer une colonne ou une table

`RENAME COLUMN … TO …` veut dire *renomme la colonne … en …*. Essaie, puis reviens au nom d'origine :

```
ALTER TABLE professeur RENAME COLUMN telephone TO gsm;
ALTER TABLE professeur RENAME COLUMN gsm TO telephone;
```

`RENAME TO` renomme la table elle-même. Là aussi, essaie, regarde l'onglet **Structure de la base**, puis reviens en arrière :

```
ALTER TABLE professeur RENAME TO enseignant;
ALTER TABLE enseignant RENAME TO professeur;
```

> ℹ️ **LES LIMITES D'ALTER TABLE AVEC SQLITE.** SQLite ne sait pas changer le type d'une colonne, ni ajouter une contrainte à une colonne qui existe, ni ajouter une nouvelle colonne `UNIQUE`. Sur une table qui contient déjà des lignes, il refuse aussi d'ajouter une colonne `NOT NULL` sans valeur par défaut, avec le message `Cannot add a NOT NULL column with default value NULL` : les lignes présentes n'auraient rien à y mettre. Quand tu modifies une colonne dans la fenêtre **Modifier la table** de DB Browser, le logiciel contourne ces limites en reconstruisant la table en coulisse.

La structure de `professeur` est maintenant complète :

```
CREATE TABLE professeur (
    id        INTEGER PRIMARY KEY,
    nom       TEXT    NOT NULL,
    prenom    TEXT    NOT NULL,
```

```
    email     TEXT    NOT NULL UNIQUE,
    actif     INTEGER NOT NULL DEFAULT 1,
    telephone TEXT
)
```

Garder ses instructions : le script SQL

Tout ce travail tient en quelques lignes de SQL. Si tu les gardes dans un fichier, tu peux recréer la même table sur un autre ordinateur, l'envoyer à quelqu'un, ou recommencer après une erreur.

Efface le contenu de la zone de requête et écris :

```sql
DROP TABLE IF EXISTS professeur;

CREATE TABLE professeur (
    id        INTEGER PRIMARY KEY,
    nom       TEXT    NOT NULL,
    prenom    TEXT    NOT NULL,
    email     TEXT    NOT NULL UNIQUE,
    actif     INTEGER NOT NULL DEFAULT 1,
    telephone TEXT
);
```

Enregistre ce texte avec le bouton d'enregistrement de l'onglet **Exécuter le SQL**, sous le nom `creation-professeur.sql`.

> 📖 **NOUVELLE NOTION : LE SCRIPT SQL**
>
> Un **script SQL** est un fichier texte, d'extension `.sql`, qui contient une suite d'instructions SQL. On peut l'ouvrir, le relire, le corriger et l'exécuter à nouveau. Il ne contient pas la base : il contient de quoi la construire.
>
> En anglais : GB ***SQL script***.

Exécute le script deux fois de suite. Aucune erreur : `DROP TABLE IF EXISTS` supprime l'ancienne table si elle existe, puis `CREATE TABLE` en recrée une vide. On dit que le script est **rejouable**.

> ⚠️ **REJOUER CE SCRIPT EFFACE TOUTES LES LIGNES DE `PROFESSEUR`.** Pendant la conception, c'est pratique : tu corriges le `CREATE TABLE` et tu relances, sans te soucier de l'état précédent. Mais sur la base réellement utilisée par l'école (on parle de base « en production »), ce même script ferait disparaître tous les professeurs. Un script qui commence par `DROP TABLE` ne s'exécute jamais sur des données réelles.

Tu as maintenant deux fichiers différents :

Fichier	Ce qu'il contient
`mon-ecole-eleves.db`	la base elle-même : ses tables et leurs lignes
`creation-professeur.sql`	des instructions capables de recréer la table `professeur`

Écrire les modifications dans le fichier

DB Browser garde tes changements en attente tant que tu ne les enregistres pas. Clique sur **Écrire les modifications** (ou `Ctrl+S`) : la table `professeur`, vide et complète, est désormais enregistrée dans le fichier `.db`.

🧪 Exercices

Exercice 1 — Lire et corriger un CREATE TABLE ★☆☆☆☆

Pour chaque instruction : exécute-la, lis le message de SQLite, trouve l'erreur et écris la version correcte.

```sql
-- a)
CREATE TABLE local (
    code  TEXT,
    etage INTEGER,
);

-- b)
CREATE TABLE local (
    code  TEXT NOT NULL
    etage INTEGER
);

-- c)
CREATE TABLE local (
    code  TEXT NOT NULL,
    etage INTEGER
;
```

La dernière est plus sournoise : elle ne provoque **aucune** erreur. Exécute-la, puis explique pourquoi elle est pire que les trois autres.

```sql
-- d)
CREATE TABLE salle (
    code  TEXT NUL,
    etage INTEGER
);
```

Supprime les tables d'essai avec `DROP TABLE IF EXISTS` quand tu as terminé.

Exercice 2 — Choisir le bon type ★★☆☆☆

Choisis `TEXT`, `INTEGER` ou `REAL` pour chaque information, et justifie en une phrase.

Information	Type
le numéro de téléphone d'un parent	
le nombre de places d'un local	

Information	Type
le prix d'un repas à la cantine	
le code postal d'une commune	
la date d'inscription d'un élève	
la moyenne d'un élève sur 20	
le numéro de compte bancaire de l'école	

Exercice 3 — Obligatoire, unique, par défaut ? ★★☆☆☆

Pour une table `livre` de la bibliothèque de l'école, indique les contraintes que tu poserais sur chaque colonne. Il n'y a pas toujours une seule bonne réponse : l'important est de justifier.

Colonne	`NOT NULL` ?	`UNIQUE` ?	`DEFAULT` ?
`titre`			
`auteur`			
`isbn` (le numéro international du livre)			
`nombre_pages`			
`disponible` (`1` ou `0`)			
`remarque`			

Exercice 4 — Écrire la table des locaux ★★★☆☆

L'école veut recenser ses locaux :

> « Chaque local a un code, comme `B14` , obligatoire et jamais en double. On note son étage, et son nombre de places, qui est obligatoire. On veut savoir s'il a un projecteur : par défaut, on considère que non. »

Écris le `CREATE TABLE local`, avec une clé primaire `id`, les bons types et les bonnes contraintes. Il doit s'exécuter du premier coup.

Exercice 5 — Faire évoluer la table ★★★☆☆

À partir de ta table `local` :

1. Ajoute une colonne `remarque` de type `TEXT`.
2. Renomme la colonne `places` en `nb_places`.
3. Sur une table `local` qui contient déjà des locaux, l'instruction `ALTER TABLE local ADD COLUMN batiment TEXT NOT NULL;` est refusée avec le message `Cannot add a NOT NULL column with default value NULL`. Explique pourquoi SQLite refuse.
4. Réécris cette instruction pour qu'elle soit acceptée même quand la table contient des lignes, en gardant `NOT NULL`.

Exercice 6 — Le script rejouable ★★★★☆

1. Écris un script `creation-locaux.sql` qui supprime la table `local` si elle existe, puis la recrée.
2. Exécute-le deux fois : vérifie qu'il ne produit aucune erreur.
3. Ajoute une ligne à la main dans l'onglet **Parcourir les données**, puis réexécute le script. Qu'est devenue la ligne ?
4. Explique en deux ou trois phrases pourquoi ce script est pratique pendant la conception et dangereux sur une base en production.

✍️ À retenir

- Le **DDL** définit la structure (`CREATE`, `ALTER`, `DROP`) ; le **DML** manipule les lignes (`SELECT`, `INSERT`, `UPDATE`, `DELETE`).
- Dans un `CREATE TABLE`, chaque colonne a un nom, un type et d'éventuelles contraintes ; **une virgule entre les colonnes, jamais après la dernière**.
- Une donnée faite de chiffres n'est pas forcément un nombre : un téléphone est du `TEXT`.
- `INTEGER PRIMARY KEY` se remplit tout seul, sans `AUTOINCREMENT`.
- Les contraintes `NOT NULL`, `UNIQUE` et `DEFAULT` font respecter des règles par la base elle-même.
- `ALTER TABLE` modifie une table sans perdre ses lignes ; `DROP TABLE` la détruit avec toutes ses lignes.
- Un script SQL rejouable est pratique en conception et dangereux en production.

Suite

La table `professeur` est prête, mais vide. L'article Ajouter des lignes : INSERT la remplit, et te montre ce que les contraintes refusent.